New Techniques for Higher-Order Superposition

Petar Vukmirović[1][0000−0001−7049−6847],
Alexander Bentkamp[1][0000−0002−7158−3595],
Jasmin Blanchette[1,2,3][0000−0002−8367−0936],
Simon Cruanes[4][0000−0003−3969−5850], Sophie Tourret[2][0000−0002−6070−796X]
, and Visa Nummelin[1][0000−0003−0078−790X]

[1] Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
{p.vukmirovic,a.bentkamp,j.c.blanchette,visa.nummelin}@vu.nl
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
stourret@mpi-inf.mpg.de
[3] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
[4] Aesthetic Integration, Austin, Texas, USA simon@aestheticintegration.com

Abstract. Even though it is the most successful calculus for first-order logic, superposition has only recently been extended to higher-order logic. Higher-order superposition introduces new challenges such as infinitely branching inference rules, new possibilities such as native reasoning with formulas, and the need for new heuristics to curb the explosion of additional higher-order rules. In this paper, we describe techniques that address all three subjects. Our techniques are implemented in Zipperposition 2 theorem prover. Both our evaluation and theorem proving competition results show that new techniques are state-of-the-art in automatic higher-order reasoning.

Keywords: higher-order, superposition, theorem proving

1 Introduction

In the last decades, first-order superposition-based automatic theorem provers have emerged as efficient and useful reasoning tools. The efficiency of superposition provers is witnessed by their dominance at annual CASC [31] theorem prover competitions, in which a superposition-based prover has been the winner of the first-order division since the competition's inception. They are also routinely used as backends to proof assistants [8, 17, 23], automatic higher-order theorem provers [29] and software verifiers [9]. Superposition technique has only recently been extended to higher-order logic by means of λ-superposition [2] and SKBCI-superposition calculi [5]. At CASC-J10[5], Zipperposition 2, higher-order prover primarily based on λ-superposition, has won the higher-order division, ending the long winning streak of tableaux-based Satallax [7].

Even though the core theory behind λ-superposition has been discussed in detail, many aspects of its successful use in practice have not been described.

[5] http://www.tptp.org/CASC/J10/

Furthermore, in higher-order logic, tableaux based approaches have some clear advantages over superposition (or more generally, resolution) based counterparts. We show how the complexity and explosiveness of λ-superposition can be kept under control in practice, and we also discuss how some successful tableaux techniques can be simulated in saturation based prover.

The main drawback of λ-superposition compared to SKBCl-superposition is the presence of rules with infinitely many conclusions. We describe a mechanism that allows to interleave obtaining conclusions for infinitely branching inferences with standard saturation process (Section 3). Our mechanism allows the prover to exhibit the same behavior as first-order superpostion on purely first order problems, smoothly scaling with increased number of higher-order clauses. We also give some heuristics to curb the explosion induced by explosive rules not present in standard superposition (Section 4).

Tableaux-based techniques take a more holistic view at a higher-order problem, as they work on the formula, rather than the clause level. We show how clausification can be tightly integrated with saturation process, as well as how useful heuristic information for higher-order proof search can be obtained form input formulas (Section 5). Collaboration with external first-order provers is traditionally used in higher-order theorem proving. As λ-superposition gracefully generalizes first-order superposition, spending much time in the backend communication can be counterproductive. We describe how to achieve a balance between allowing the higher-order calculus to do native reasoning and handing off reasoning to more efficient first-order backend (Section 6). Last, interesting patterns can be observed in various higher-order encodings of problems. We show how we can use those patterns to make higher-order reasoning easier (Section 7).

All of the described techniques are implemented in Zipperposition 2. At CASC-J10, Zipperposition 2 solved 84% of given higher-order problems, which is 20% percentage points ahead of the next best prover, Satallax 3.4. To perform a more extensive evaluation, we compare Zipperposition 2 with other provers on all monomorphic higher-order TPTP benchmarks [32] (Section 8). Zipperposition 2 still comes out as a winner, solving 77 problems more than the next best prover.

2 Background

Our setting is that of simply typed higher-order logic. Following our earlier work [35], we define terms as either free variables F, G, X, Y, bound variables x, y, z, constants $\mathsf{f}, \mathsf{g}, \mathsf{a}, \mathsf{b}$, applications $s\,t$ or λ-abstractions $\lambda x.\, s$. The syntactic distinction between free and bound variables introduces *loose bound variables* (bound variables with no enclosing λ-binder, as y is in $\lambda x.\, y\,\mathsf{a}$), but it increases clarity. We shorten iterated application $(s\,t_1)\cdots t_n$ to $s\,\bar{t}_n$, and λ-abstraction $\lambda x_1.\cdots\lambda x_n.\, s$ to $\lambda\overline{x}_n.\, s$. Every β-normal term s can be written as $\lambda\overline{x}_m.\, h\,\bar{t}_n$, where h is not an application; we call h the *head* of the term. If the head is a variable we call s a *variable-headed term*; if $n > 0$ we call s an *applied variable*; if h's return type is Boolean, we call h a *predicate variable*. A literal l is an equation $s \approx t$ or a disequation $s \not\approx t$. Clause is a multiset of literals, interpreted

disjunctively. Non-equational (predicate) literals, such as $\mathsf{even}(x)$ are encoded as $\mathsf{even}(x) \approx \top$ (or as $\mathsf{even}(x) \approx \bot$ if they are negative).

We are concerned with improving practical applicability of λ-superposition calculus in whose development all authors except for Cruanes and Nummelin were involved [2]. This is a complete calculus for Boolean-free extensional polymorphic clausal higher-order logic. Vukmirović and Nummelin describe a pragmatic approach to extending λ-superposition to higher-order logic with Booleans [24]. This calculus makes extensive use of *complete set of unifiers (CSU)*. CSU for s and t, denoted $CSU(s,t)$ is a set of unifiers such that for any unifier ϱ of s and t, there is $\sigma \in CSU(s,t)$ such that $\varrho = \sigma \circ \theta$, for some substitution θ . It is a well known fact that CSUs can be infinite, making base rules of λ-superposition such as superposition or equality resolution infinitely branching.

An alternative to infinitely branching rules of λ-superposition is SKBCI-superposition calculus [5] which uses a form of first-order unification but enumerates higher-order terms using rules that instantiate applied variables with partially applied combinators from the complete combinator set $\mathsf{S}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{I}$. This calculus is the basis of Vampire 4.5 which was the second-best superposition based theorem prover in higher-order division of CASC-J10.

Zipperposition 2 [?] is a higher-order theorem prover based on a pragmatic extension of λ-superposition. It was conceived as a testbed for rapid experimenting with extensions of first-order superposition, but over time, it has assimilated most of the techniques and heuristics of the state-of-the-art superposition prover E [26]. Zipperposition 2 also implements SKBCI-superposition and some of the techniques we describe are also applicable to this calculus.

Many of our techniques are concerned with extending *given-clause* loop, the standard saturation algorithm, to support higher-order reasoning patterns. Given-clause loop splits the proof state in the set P of processed clauses (initially empty), and the set U of unprocessed clauses (initially containing all clauses). At each iteration of given-clause loop, a clause C from U is moved to P, all inferences between C and clauses in P are performed and the conclusions are stored in U.

We have implemented all of the described techniques in Zipperposition 2. To asses usefulness of the techniques, we carried out some experiments with this implementation. All of the experiments use 2606 monomorphic higher-order TPTP [32] v7.2.0 problems as the benchmark set. We fixed a *base* configuration of Zipperposition parameters as a baseline for all comparisons. Then, in each experiment parameters that are concerned with a particular technique are varied to evaluate usefulness of the technique. Experiments are performed on StarExec [30] servers, which are equipped with Intel Xeon E5-2609 0 CPUs clocked at 2.40 GHz. Unless otherwise stated, we used CPU time limit of 20 seconds, which is roughly the time a single configuration is given in portfolio mode.

3 Dealing with Infinitely Branching Inferences

The Technique Integrating inference rules that require enumeration of a CSU with the standard given-clause algorithm is both theoretically [37] and practi-

cally [28, Section 4.3.2.] challenging issue. As an illustration, consider higher-order resolution between clause $C \vee s \approx \top$ and $D \vee t \approx \bot$ which possibly yields infinitely many resolvents of the form $(C \vee D)\sigma$ where $\sigma \in CSU(s,t)$. Theorem prover that uses such a rule must implement a mechanism that fairly enumerates all such resolvents between all eligible clauses. We describe how to modify the given-clause loop to allow enumeration of conclusions of inferences that rely on elements of a CSU.

The cornerstone of a fair enumeration of inference conclusions is the ability to evaluate a short prefix of the infinite sequence, and save the remaining of the sequence for a later visit. However, as the existence of the *next* element of the CSU is undecidable, it is possible that the request for the next inference conclusion will not terminate, brining the theorem prover to a halt. Thus, we require that the procedure for CSU enumeration does not simply return a lazily evaluated stream [20] [22, Section 4.2] of unifiers, but to also periodically return a special *back-off value* N which signifies that control should be returned to the caller of the unification procedure. One such procedure is the complete unification procedure of Vukmirović et al. [35], but other standard unification procedures [14, 15] can be modified to fulfill this requirement. Based on the stream of unifiers interspersed with N, we can construct a stream of inferences (similarly interspersed with N) for which finite prefixes of any size can be computed without the concern that this computation will be non-terminating.

Each inference stream is assigned a priority based on the kind and premises of an inference, and it is stored in the priority queue Q. For conclusions computed from streams in Q to be considered in the given-clause loop, they should eventually be moved to the passive clause set. In each iteration of the given-clause loop, we take at most n clauses from Q as follows: we remove a stream s with the highest priority from Q and evaluate the first element of s which can either be N or an inference conclusion c. If the stream yields N, the same stream will be probed for a clause up to k times. After we either obtain the clause or run out of probe possibilities, the unevaluated part of the stream s, denoted s', is given a decreased priority based on the number of times an ancestor of s was chosen for evaluation and whether a clause was obtained from s or not. Finally, s' is stored in Q using the newly assigned priority. This process is repeated n times. Whenever a stream becomes empty, it is removed from Q.

This process does not guarantee that each stream will eventually be visited, compromising the fairness of the saturation loop. Thus, we must periodically try taking a clause from i streams that were first added to Q, where i is initially 1 and increases with each attempt to take clauses from streams fairly. Similarly, when the set of passive clauses is empty, we will repeatedly visit all streams until Q becomes empty, or until we obtain a clause from any stream.

Even though the described mechanism achieves the goal of guaranteeing that each conclusion of rules with infinitely many conclusions will be considered, there is still place for improvement. Unification procedure by Vukmirović et al. terminates on many fragments of higher-order terms including first-order and pattern fragment [21]. If none of these simple fragment algorithms finds a

unifier, the procedure immediately yields N before continuing the search. Thus, forcing computation of at most one conclusion based on the returned stream of unifiers is guaranteed to be efficient. If computing the first element of the stream yields a unifier, the corresponding conclusion is directly put in the passive set. The rest of the inference stream is stored in Q. This allows us to keep the same behavior as first-order resolution (or superposition) prover on first-order problems as conclusions with first-order most general unifiers will be immediately stored in the passive set. Furthermore, on a problem containing mostly first-order clauses, Q will not be visited to compute the results of first-order inferences.

The design of this mechanism was guided by folklore knowledge about higher-order theorem proving. First, it is widely accepted [**?**] that most steps even in long higher-order proofs are between first-order clauses. The unification algorithm and inference scheduling system ensure that first-order inference conclusions are put in the proof state as early as possible. Second, some inference rules are expected to be less useful than the others. We initialize the penalty of the stream differently for each inference, allowing old streams of more useful inferences to be queried before newly added, but potentially less useful stream. Last, if we use a unification algorithm that has aggressive redundancy elimination, we will often find the necessary unifier within the first few unifiers returned. Similarly, if a stream keeps returning N, it is likely that it is blocked in a non-terminating computation and should be ignored. Heuristics that increase the stream penalties take into account both observations.

Discussion and Evaluation We have implemented the unification procedure by Vukmirović et al. in Zipperposition, and integrated it with the described inference scheduling mechanism to enumerate conclusions of λ-superposition inference rules. Zipperposition is the only competing higher-order prover that proves all Church numeral problems from the TPTP library very efficiently [35]. These are hard unification problems, on which the stream system allows the prover to explore the proof state lazily. Consider the TPTP problem `NUM800^1`, which claims that there exists a function F such that $F\,\mathsf{c}_1\,\mathsf{c}_2 \approx \mathsf{c}_2 \wedge F\,\mathsf{c}_2\,\mathsf{c}_3 \approx \mathsf{c}_6$, where c_n stands for n-th Church numeral $\mathsf{c}_n = \lambda s\,z.\,s^n(z)$ (expontent denotes iterated application). This problem is clearly solved if F is instantiated with the multiplier operator $\lambda x\,y\,s\,z.\,x\,(y\,s)\,z$. However, this unifier is only one out of many available for each occurrence of F. Provers that limit themselves to only finitely many unifiers face a difficult heuristic choice: impose a shallow unification depth and possibly miss the unifier, or deepen the unification and a spend long time in the unification procedure. The time spent in the unification procedure can be so long that no clauses get created. In an independent evaluation setup, Vukmirović et al. [35, Section 7] compare complete and depth-limited (pragmatic) variant of the unification procedure. Pragmatic version of the procedure is used directly – all the inference results are put in the passive set. Interestingly, even though it is much more complex, the complete variant solves only 15 problems less that the most competitive pragmatic version. Moreover, it solves 19 problems not solved by the pragmatic version. We believe that the use of stream system made com-

plete version more competitive and allowed it to defer exploring less promising branches of the unification tree.

Out of competing higher-order theorem provers, to the best of our knowledge, only Satallax uses infinitely branching calculus rules. It maintains a queue of *commands* that contain instructions how to create a successor state in the tableaux. One of the commands describes infinite enumeration of all closed terms of a given function type. Each execution of this command makes progress in the said enumeration. The principal difference between commands and streams is that each execution of a command is guaranteed to make progress in enumerating the next closed functional term. On the other hand, existence of the next unifier in the stream is an undecidable problem and periodically returning the control to the prover is the key to making progress.

4 Limiting the Explosiveness of Prolific Rules

The Technique To accommodate features of higher-order logic such as function extensionality and quantification over functions, many complete calculi employ rules for which most clauses can be used as the premises. For example, λ-superposition extends first-order superposition with a rule FluidSup [2], which almost always applies to two clauses if one of them contains a term of the form $F\,\overline{s}_n$. To keep rules like these under control we describe three mechanisms for limiting their applicability:

- Penalize streams of expensive inferences
- Defer selecting the resulting clauses for processing
- Restrict applicability of the rules

Complete provers that implement λ-superposition, like Zipperposition, have two main heuristic choices: which inference stream to query for clauses and which clauses from the passive set to choose for processing. First and second mechanisms allow to control those two choice points. Third mechanism sacrifices completeness by applying the inference rules only to a subset of chosen clauses.

We propose the following scheme for penalization of inference streams: penalty of each stream is given an initial value based on properties of inference premises such as their derivation depth. For prolific rules, like FluidSup, we increment this value by a parameter p. Penalties for less prolific rules, such as DupSup are increased by fraction of p, $\lfloor \frac{p}{3} \rfloor$ for example.

Most saturation provers organize the passive clause set into a queue which uses heuristic clause weights as keys. Inspired by E [26], Zipperposition visits multiple such queues in a round-robin fashion, and uses a pair of two numbers called *clause priority* and *clause weight* as the queue key. Similar layered approach has recently been used in Vampire [11]. Clause priority gives crude separation of clauses into priority classes which are further ordered by the clause weight. To isolate clauses which were obtained using higher-order inferences, we implemented priority function `PreferHOSteps` which assigns higher

base	`BAVN`	`PL`	`PSAV`	`PHOS`	`BNF`	`PDAV`
1636 (0)	1638 (2)	1638 (14)	1635 (0)	1630 (2)	1594 (2)	1518 (0)

Fig. 1: Effect of priority function on success rate

base	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
1636 (16)	1609 (0)	1612 (1)	1617 (0)	1621 (2)	1618 (0)

Fig. 2: Effect of FluidSup penalty on success rate

priority if an inference rule that is not part of standard first-order superposition was used in the clause derivation. Easy to compute overapproximation of this function, `PreferLambdas` prefers clauses that have λ-abstraction subterms. Another new priority function we introduced is `ByNormalizationFactor`, which is based on the observation that a higher-order inference that applies complicated substitution to a clause is usually followed by a $\beta\eta$-normalization step. If $\beta\eta$-normalization greatly reduced the size of a clause, it is likely that this substitution simplified the clause (e.g., by removing arguments of a variable). Thus, this function prefers clauses which are results of $\beta\eta$-normalization, and within those clauses prefers the ones with larger size reductions. We have also implemented a priority function `PreferShallowAppVars` which prefers clauses with lower depths of the deepest occurrence of an applied variable. This function tries curbing the explosion of both λ- and SKBCI-superposition. Intuitively, applying substitution to a top-level applied variable often reduces this applied variable to a term with a constant head, which likely results in a less explosive clause. We have also implemented functions `PreferDeepAppVars`, which returns the priority opposite of `PreferShallowAppVars` and `ByAppVarNum` which prefers clauses with less occurrences of applied variables.

We have observed that in practice it suffices to apply prolific higher-order rules only to initial clauses or clauses with shallow derivation depth. To that end, we forbid application of a rule if derivation depth of any premise exceeds the limit.

Discussion and Evaluation In the base configuration, Zipperposition visits a queue with constant priority. To evalaute different priority functions, we changed the constant priority with the priorities described above. We report the results in Figure 1, where columns denote different priority functions, and are labelled using the uppercase letters in function names. To test if different queue priorities make a difference on success rate, we enabled FluidSup (which is disabled in base) and used different parameters p for priority increment of FluidSup inference queues. Results are shown in Figure 2. We have also tested limiting applicability of some rules to shallow clauses, but the results were too similar to make any interesting remark. In all of the figures, numbers in cells are of the form $a\,(b)$ where a is the number of proved problems, and b is the number of problems proved only by given configuration and no other configuration in the same figure.

Figure 1 shows that success of priority functions that require inspecting the proof of the clause (like `PHO` and `BNF`) does not justify their computational complexity: they decrease overall success rate and result in only 4 unique solutions. Simple functions like `PL` are more effective. Number of applied variables seems to be better predictor of clause usefulness than the depth of an apllied variable.

As expected, giving low penalty to FluidSup is detrimental to performance. However, as the column for $p = 16$ in Figure 2 shows, we should not give too high penalty increase either, since that avoids useful FluidSup inferences.

5 Native Reasoning with Formulas

The Technique Higher-order logic erases the difference between terms and formulas, which is present in first-order logic. In many cases, we can easily find the proof if we instantiate a predicate variable with a particular formula. Finding this formula is usually easier if the problem is not clausified, as the example $\exists F.\, F\,\mathsf{p}\,\mathsf{q} \Leftrightarrow \mathsf{p} \wedge \mathsf{q}$ shows. This form of the problem makes finding the correct instantiation $\lambda x\, y.\, x \wedge y$ easier, whereas the negated and clausified form $F\,\mathsf{p}\,\mathsf{q} \approx \bot \vee \mathsf{p} \approx \top, F\,\mathsf{p}\,\mathsf{q} \approx \bot \vee \mathsf{q} \approx \top$, $F\,\mathsf{p}\,\mathsf{q} \approx \top \vee \mathsf{q} \approx \bot \vee \mathsf{p} \approx \bot$ looses structural information which can be used as a heuristic for finding the correct instantiation. One of the strengths of tableaux-based provers such as Satallax is that they do not clausify the input problem which can explain Satallax' dominance on THF division of CASC competitions until CASC-J10.

Usual workflow of a resolution (or superposition) based theorem prover is to clausify the input problem, simplify the initial clause set and start the saturation loop. As mentioned example shows, for higher-order logic clausification can cause loss of important structural information. Thus, we propose alternative workflow which integrates clausification tightly in the saturation loop.

Instead of clausifying the initial problem, we represent input formula f as a higher-order clause $f \approx \top$. We then extend the λ-superposition calculus with *lazy clausification* rules [24, Section 3.4]. These incrementally clausify top-level logical symbols: clause $C' \vee (p \wedge q) \approx \bot$ yields a new clause $C' \vee p \approx \bot \vee q \approx \bot$.

Lazy clausification rules can be used as inference rules, in which case new clauses are added to the passive set, or as simplification rules which replace the premise with conclusions. The former gives more flexibility as all intermediate states in clausification will be stored in the proof state, at the cost of producing many clauses. The latter produces less clauses while it still allows the prover to analyze the syntactic structure of formulas. Following folklore knowledge that clausifying equivalences is especially detrimental to preserving syntactic structure [10], we use rules that clausify (dis)equivalence only as inferences.

Integration of clausification with the proving process has many merits. First, during saturation, prover might derive unit clauses which can be used to demodulate any subterm in the unclausified part of of the problem. This kind of simplification is impossible to simulate as a preprocessing step in the usual workflow.

Second, if we use lazy clausification as an inference, base (λ-superposition) inferences can be performed between formulas rather than clauses, potentially

shortening proofs [24]. This is especially useful for the example we discussed, which can be solved by negating the conjecture, eliminating the universal quantifier and using equality resolution to resolve the resulting literal $F\,\mathsf{p}\,\mathsf{q} \not\approx \mathsf{p} \wedge \mathsf{q}$.

Third, some superposition provers support variants of AVATAR [34], a technique that allows to split clauses into free variable disjoint parts and continue the proof search in smaller fragments of the search space. Combining AVATAR and native formula reasoning using lazy clausification allows us to split a formula f into variable disjoint parts $f_1 \vee \cdots \vee f_n$. If we are able to derive $\bot$ under each of the assumptions f_i, we have finished the proof attempt. This technique is surprisingly similar to tableaux-based methods, but allows us to use strengths of superposition such as powerful redundancy criterion and aggressive rewriting to close each of the "branches" more efficiently than tableaux-based methods.

Last, even when lazy clausification is used as a simplification rule, the prover can use the information obtained from the formulas on which clausification rules are applied to make heuristic choices. In particular, each clause chosen for processing is scanned for λ-abstractions that return Boolean type and found terms are stored in the set *Inst*. Optionally, *Inst* can contain *primitive instantiations* [24], that is, imitations of logical symbols which approximate the shape of a formula that a predicate variable can be instantiated with. We have also observed that adding abstractions of the literals in the conjecture provides useful instantiations for the induction axioms. For example, we can abstract a in $\mathsf{a}{+}(\mathsf{b}{+}\mathsf{c}) \not\approx (\mathsf{a}{+}\mathsf{b}){+}\mathsf{c}$ as $\lambda x.\, x + (\mathsf{b} + \mathsf{c}) \not\approx (x + \mathsf{b}) + \mathsf{c}$ and add this abstraction to *Inst* as a trigger for instantiating predicate variable in the induction axiom. Found triggers are used in conjunction with lazy clausification as follows: whenever a bound variable x is replaced by a fresh free variable Y in a clause of the form $C = C' \vee \forall x.\, f \approx \top$ (or $C = C' \vee \exists x.\, f \approx \bot$), we additionally replace x with all terms $t \in \mathit{Inst}$ that are of the same type as x. However, as new triggers t can be discovered after eligible clause C has been processed, we also save all clauses C for which universally bound predicate variable has been eliminated. When a trigger is discovered, we instantiate all eligible saved clauses. Note that, unlike in first-order superposition, such instantiations usually change the clause enough that standard subsumption algorithm will not be able to recognize instances as redundant and remove them from the proof state, undoing mentioned instantiation.

Discussion and Evaluation Base configuration uses standard clausification, and disables simplified version of AVATAR, implemented in Zipperposition [**?**]. To test the first three discussed merits, we change *base* along two axes: we choose either standard clausification (`SC`), lazy clausification as inference (`LCI`) and lazy clausification as simplification (`LCS`) as the clausification method; for each clausification method we enable (+`A`) or disable (−`A`) simplified AVATAR.

As observed from results in Figure 3, interleaving simplification techniques with clausification increases success rate by a large margin. On the other hand, using lazy clausification as an inference, decreases success rate by larger margin. Overall, simplified version of AVATAR causes performance loss, but helps in corner cases, as the number of uniquely solved problems shows.

	+A	−A
SC	1624 (21)	1636 (33)
LCI	1496 (3)	1532 (6)
LCS	1657 (5)	1709 (21)

Fig. 3: Interaction of AVATAR and clausification method

We have also tested enabling Boolean instantiation in LCS mode without AVATAR which resulted in proving 1743 problems and 36 unique solutions, with respect to Figure 3. This suggests that finding more ways to interleave saturation and instantiation might be beneficial for higher order theorem proving. Boolean instantiation techniques help us solve problems that require inductive reasoning on datatypes (such as DAT056^2) or the ones that are better suited for tableaux (such as SEU868^5 [7]).

6 Collaboration with Backend System

The Technique Cooperation with efficient off-the-shelf first-order theorem provers is an essential feature of higher-order theorem provers such as Leo-III [28, Section 4.4], Satallax [7] and Sledgehammer [6]. Those provers invoke first-order backends many times during a proof attempt and spend significant time translating higher-order formulas to first-order logic. For provers based on modern higher-order calculi, like λ-superposition, which are more efficient on first-order problems, such aggressive collaboration might be counterproductive. As these calculi will likely efficiently derive useful first-order conclusions themselves, the time spent invoking an external prover might be more effectively spent doing actual native higher-order reasoning. We propose several limitations to the way external provers are currently used in an attempt to achieve a balance between letting native higher-order reasoner explore the search space and handing off the control to more efficient, but less powerful backends.

In his thesis [28, Section 6.1], Alexander Steen extensively evaluates effects of using different first-order backends to the success rate of Leo-III. His results suggest that using a different first-order backend makes much weaker effect than the use of any backend compared to only native Leo-III reasoning. Following these results, and since we want to limit the communication with backends, we focus our attention to using a single backend. The backend we focus on is Ehoh [36], an extension of E [26], which is a state-of-the-art superposition prover with syntactic support for higher-order logic features such as partial application, applied variables, and formulas occurring as arguments of function symbols. Use of Ehoh provides efficiency of E, while making translation from full higher-order logic much easier: the only feature of higher-order logic that is not syntactically supported is λ-abstraction. However, Ehoh's higher-order reasoning capabilities are limited as it is unable to synthesize λ-abstractions, essentially uses first-order unification, and does not perform any predicate variable instantiation.

We also severely constrain the way in which the backend is invoked. Instead of regularly invoking the backend, we allow it to be invoked at most once in a single proof attempt. The reasoning behind this decision is that most competitive higher-order provers use a portfolio mode in which many configurations are ran for a short time. Invoking a backend many times during a short run of a configuration would likely leave little time for native higher-order reasoning. Furthermore, superposition-based backends cannot be used incrementally, meaning that each invocation will start with no knowledge of the previous ones.

Choice of clauses to translate for is critical for finding the proof. Let M denote the union of the active and passive set at the point of given-clause loop interruption. Furthermore, let M_0 denote a subset of M containing initial clauses (i.e., clauses with derivation depth of 0), and let M_{ho} be a subset of $M \setminus M_0$ containing only clauses in whose derivation at least one inference rule tagged as higher-order is used. Rules that are tagged as higher-order are ones that are not part of first-order superposition, but also all the standard superposition rules in which applied substitution contained higher-order terms such as λ-abstractions. We order clauses in M_{ho} by their derivation depth and weight, and choose all clauses in M_0 and first n clauses from M_{ho} for use with backend reasoner. We include all the clauses in M_0 since they constitute the backbone of the initial problem. Out of derived clauses we include only clauses from M_{ho} with shallow derivation depth because clauses derived using first-order rules will also be derived using Ehoh and since we expect clauses with deep derivation depth to be less useful.

The remaining step is the translation of λ-abstractions. We support two different translation methods: either using λ-lifting [16] or Curry's SKBCI combinator translation [33] (without the corresponding combinator definition axioms, as they are notoriosly explosive [5]). Furthermore, we also support a third mode in which clauses with λ-abstractions are simply removed from the translation.

Discussion and Evaluation Zipperposition can choose many parameters of backend collaboration including the CPU time alloted for Ehoh, Ehoh's parameters, the moment of Ehoh's invocation, size of M_{ho}, and λ-abstraction translation method. Due to limited resources, we fix the first two parameters to 5 seconds and automatic mode, respectively, and evaluate the last three parameters.

Ehoh is invoked after $p \cdot t$ CPU seconds, where $p \in [0, 1)$ and t is the total CPU time alloted to Zipperposition. Figure 4 shows the effect of varying p, while the size of M_{ho} is fixed to 32 and and λs are encoded using lifting. As expected, high-performance backend such as Ehoh greatly improves the success rate of a less optimized prover such as Zipperposition. Even though the differences between invocation point are small, it seems better to invoke the backend early.

Figure 5 shows how the size of M_{ho} affects the success rate, when invocation point is fixed at 0.25 and we use λ-lifting. Including a small number of higher-order clauses with the lowest weight performs better than including a large number of such clauses, even though the latter produces some unique solutions.

In Figure 6, we show the effects of λ-abstraction translation methods, where drop denotes that clauses with this feature are removed from the translation. Invocation point is fixed at 0.25 and size of M_{ho} is set at 32. Surprisingly, re-

base	$p = 0.1$	$p = 0.25$	$p = 0.5$	$p = 0.75$
1636 (1)	1903 (9)	1897 (0)	1900 (6)	1886 (1)

Fig. 4: Effect of invocation point on success rate

base	$m = 16$	$m = 32$	$m = 64$	$m = 128$	$m = 256$	$m = 512$
1636 (2)	1896 (5)	1897 (0)	1889 (2)	1888 (2)	1886 (1)	1872 (4)

Fig. 5: Effect of size of M_{ho} (denoted with m) on success rate

moving clauses with λ abstractions performs comparable to using translations. Out of different translation methods, λ-lifting comes out as a clear winner.

7 Preprocessing Higher-Order Problems

The Technique TPTP library [32], the de-facto standard for benchmarking automatic theorem provers, contains thousands of higher-order problems. Even though these problems are significantly different, we can observe some conspicuous patterns in their encodings. Most notably, THF (higher-order) problems of the TPTP library, extensively use the `definition` annotation of the input formulas. Those annotations are reserved for universally quantified equations (or equivalences) that define symbols. In some cases, eagerly unfolding all the definitions can eliminate all higher-order features, significantly simplifying the problem. In other cases, unfolding the definitions significantly increases the size of the problem, making it very hard for a theorem prover. Therefore, higher-order prover should allow flexible treatment of definitions.

An effective way to deal with definitions is to interpret them as rewrite rules, using the orientation given in the input problem. If there are multiple definitions for the same symbol, only the first one is used as a rewrite rule. Then, whenever a clause is picked in the given-clause loop, it will be rewritten using the found rewrite rules. As there are no constraints put on the definitions, this rewriting might not terminate. Therefore, to ensure termination, it is crucial to limit the number of applied rewrite steps. In practice, we have observed that TPTP problems are usually well-behaved: only one definition per symbol is given and the definitions are not self-referential. An alternative to rewriting when clauses are chosen for processing is to rewrite the input formulas as a preprocessing step. Advantage of this eager approach is that input clauses will be fully simplified when the proving process starts, and no opaque defined symbols will occur in clauses, which disallows them to mislead the sensitive heuristics.

Treating definitions as rewrite rules might be effective in some cases, but it clearly compromises the completeness of calculi like λ-superposition, which use term orderings. One of the most successful term orderings used in superposition-based calculi is Knuth Bendix order (KBO) [18], which compares terms using symbol precedence and symbol weight. It was recently adapted for use with λ-superposition [2]. Given a symbol weight assignment $\mathcal{W}$, we can easily transform

base	lifting	SKBCI	drop
1636 (1)	1897 (53)	1864 (7)	1855 (4)

Fig. 6: Effect of λ translation method on success rate

base	RW before CNF	no RW	no RW + KBO
1636 (67)	1626 (26)	1303 (2)	1323 (15)

Fig. 7: Effect of different rewrite methods

it into $\mathcal{W}'$, such that each definition of the form $\mathsf{f}\,\overline{X}_m \approx \lambda\overline{y}_n.\,t$ where only free variables of t are $\overline{X}_m$, no free variable repeats in t, and f does not occur in t can be oriented from left to right. For each symbol f we collect all definitions of the described form. Then, we set $\mathcal{W}'(\mathsf{f}) = w+1$ where w is the maximum weights for all right-hand sides of the definitions for f, computed using $\mathcal{W}$. Now, it is easy to see that the weight of the left-hand side will be large enough to orient all definitions.

To filter out axioms that are unlikely to be useful in a proof attempt, many theorem provers use SInE algorithm [13]. In essence, this algorithm starts with the set of symbols occurring in the conjecture, and tries to find axioms that define the properties of such symbols. Then, it finds the definitions of newly found symbols until fixpoint. As presence of the axioms annotated with `definition` greatly simplifies this search, we modified SInE to optionally include definitions of symbols in the conjecture. We also discovered that many encodings use terms like $\mathsf{collect}\,(\lambda x.\,\mathsf{p}\,x \wedge \mathsf{q}\,x)$ to denote a set of elements satisfying predicates p and q. Such terms usually occur in conjecture, but also in almost all axioms. Somewhat counterintuitively, not including the definition for symbol $\mathsf{collect}$ can help prove the problem. Thus, we changed SInE to remove n most commonly occurring symbols from the set of symbols occurring in conjecture.

Discussion and Evaluation Axioms tagged with `definition` are treated as rewrite rules in *base*. In Figure 7, we additionally test effects of (from left to right) rewriting the formulas before clausification, disabling special treatment of `definition` axioms, and disabling special treatment of `definition` while using adjusted KBO weight as descibed above. Results show that special treatment of `definition` axioms greatly improves success rate. Adjusting KBO weights results in obtaining many unique solutions.

Effects of removing n most common symbols from the initial symbol set in SinE algorithm are presented in Figure 8. Base configuration does not remove any symbols. Even though the results get worse with increase of n, unique solutions are obtained for higher values of n. Base configuration includes definitions of symbols occurring in the conjecture by default. If this option is disabled, the number of solved problems drops down to 1634, but 3 solutions not found by *base* are found this way.

base	$n = 1$	$n = 2$	$n = 4$	$n = 8$	$n = 16$
1636 (2)	1634 (0)	1625 (0)	1588 (0)	1543 (2)	1397 (2)

Fig. 8: Effect of ignoring n most commonly occurring symbols

	uncoop	coop
CVC4	-	1810 (7)
Leo-III	1641 (0)	2108 (3)
Satallax	2089 (1)	2224 (6)
Vampire	-	2096 (4)
Zipperposition	2222 (0)	2301 (12)

Fig. 9: Comparison with other competing higher-order theorem provers

8 Comparison with other provers

Results shown in the previous sections suggest that different choices of parameters lead to noticeably different sets of solved problems. In an attempt to use Zipperposition 2 to its full potential, we have created a portfolio mode that runs up to 50 different configurations during the runtime. We compare Zipperposition 2 to the latest versions of all competing higher-order provers: CVC4 1.8 [1], Leo-III 1.5 [29], Satallax 3.5 [7], and Vampire 4.5 [19].

For our comparison we use the same set of 2606 monomorphic higher-order TPTP v7.2.0 problems, but we try to replicate the setup of CASC more faithfully. CASC was run on 8-core CPUs with 120 s wallclock limit and 960 s CPU limit. As we run our experiments on 4-core CPUs, we set the wallclock limit to 240 s and keep the same CPU limit.

Leo-III, Satallax and Zipperposition can use backend provers to finish the proof attempt. We also run them in uncooperative mode, to evaluate their performance when cooperation with backend is disabled. The evaluation results are shown in Figure 9.

Zipperposition has the least difference in success rate between cooperative and uncooperative modes, and its uncooperative mode is only 2 problems behind the second best prover in its cooperative mode. This confirms our intuition that higher-order superposition is a good basis for automatic higher-order reasoning. Increase in success rate due to use of efficient backend suggests that implementation of this calculus in a state-of-the-art first-order superposition prover is a reasonable goal. High number of uniquely solved problems for CVC4 and Satallax suggests that there are still some non-superposition techniques whose use in saturation-based provers needs to be investigated.

9 Discussion and Related Work

The research presented in this paper has two goals. First, we give some solutions for the issues that arise with implementation of λ-superposition. Second, we

discuss many choices that can be made during proof search with this calculus, and provide detailed evaluation for each such choice.

Extensions of first-order superposition to higher-order logic have been described in detail from the theoretical point of view [2–5]: the calculi are described, their completeness is discussed and some practical extensions are given. However, none of these calculus descriptions discusses practical aspects of higher-order superposition such as heuristics and curbing the explosion induced by explosive new rules.

For standard first-order superposition there is vast literature in which practical aspects of this calculus is discussed. The literature provides detailed evaluation of different algorithms and techniques [12,25], literal selection functions [27], and clause evaluation functions [11]. In contrast, higher-order calculi implemented in competitive provers are usually discussed from a theoretical viewpoint.

10 Conclusion and Future Work

We have described some ways in which explosion incurred by λ-superposition calculus can be kept under control. First, we designed a system which allows to enumerate infinite sets of higher-order inference conclusions, while maintaining same behavior as first-order superposition on first-order clauses. Second, for many choices that can be made during higher-order proof search we describe reasonable heuristics and give their detailed evaluation. Last, we show how native support for Boolean type, brought by higher-order logic, can be used to alter the usual workflow used by saturation provers and increase the success rate. Given techniques and heuristics are implemented in Zipperposition 2 theorem prover and are shown to be state-of-the-art in higher-order reasoning.

We plan to implement all mentioned techniques and heuristics in Ehoh. As Ehoh is based on one of the best-performing first-order automatic theorem provers, we expect this implementation to perform even better than Zipperposition 2. Furthermore, we plan to investigate the proofs found by provers like CVC4 and Satallax, but missed by Zipperposition. Finding the exact reason behind why Zipperposition failed to produce proof will likely result in some new techniques or heuristics.

Acknowledgment We are grateful to the maintainers of StarExec for letting us use their service. European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Nummelin has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

1. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011)

2. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. In: CADE. Lecture Notes in Computer Science, vol. 11716, pp. 55–73. Springer (2019)
3. Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: IJCAR. Lecture Notes in Computer Science, vol. 10900, pp. 28–46. Springer (2018)
4. Bhayat, A., Reger, G.: Restricted combinatory unification. In: CADE. Lecture Notes in Computer Science, vol. 11716, pp. 74–93. Springer (2019)
5. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: IJCAR (1). Lecture Notes in Computer Science, vol. 12166, pp. 278–296. Springer (2020)
6. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: IJCAR. Lecture Notes in Computer Science, vol. 6173, pp. 107–121. Springer (2010)
7. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. J. Autom. Reason. **51**(1), 57–77 (2013)
8. Czajka, L., Kaliszyk, C.: Hammer for coq: Automation for dependent type theory. J. Autom. Reason. **61**(1-4), 423–453 (2018)
9. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
10. Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. In: CADE. Lecture Notes in Computer Science, vol. 2741, pp. 335–349. Springer (2003)
11. Gleiss, B., Suda, M.: Layered clause selection for theory reasoning. CoRR **abs/2001.09705** (2020)
12. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: KI. Lecture Notes in Computer Science, vol. 5803, pp. 435–443. Springer (2009)
13. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: CADE. Lecture Notes in Computer Science, vol. 6803, pp. 299–314. Springer (2011)
14. Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975)
15. Jensen, D.C., Pietrzykowski, T.: Mechanizing *omega*-order type theory through unification. Theor. Comput. Sci. **3**(2), 123–171 (1976)
16. Johnsson, T.: Lambda lifting: Treansforming programs to recursive equations. In: FPCA. Lecture Notes in Computer Science, vol. 201, pp. 190–203. Springer (1985)
17. Kaliszyk, C., Urban, J.: Hol(y)hammer: Online ATP service for HOL light. Math. Comput. Sci. **9**(1), 5–22 (2015)
18. KNUTH, D.E., BENDIX, P.B.: Simple word problems in universal algebras. In: LEECH, J. (ed.) Computational Problems in Abstract Algebra, pp. 263 – 297. Pergamon (1970)
19. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 1–35. Springer (2013)
20. Landin, P.J.: Correspondence between ALGOL 60 and church's lambda-notation: part I. Commun. ACM **8**(2), 89–101 (1965)
21. Nipkow, T.: Functional unification of higher-order patterns. In: LICS. pp. 64–74. IEEE Computer Society (1993)
22. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
23. Paulson, L.C., Blanchette, J.C.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: IWIL@LPAR. EPiC Series in Computing, vol. 2, pp. 1–11. EasyChair (2010)

24. Petar Vukmirović, V.N.: Boolean reasoning in a higher-order superposition prover. In: PAAR (2020)
25. Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: CADE. Lecture Notes in Computer Science, vol. 9195, pp. 399–415. Springer (2015)
26. Schulz, S., Cruanes, S., Vukmirovic, P.: Faster, higher, stronger: E 2.3. In: CADE. Lecture Notes in Computer Science, vol. 11716, pp. 495–507. Springer (2019)
27. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: IJCAR. Lecture Notes in Computer Science, vol. 9706, pp. 330–345. Springer (2016)
28. Steen, A.: Extensional Paramodulation for Higher-order Logic and Its Effective Implementation Leo-III. Dissertationen zur künstlichen Intelligenz, Akademische Verlagsgesellschaft AKA GmbH (2018), https://books.google.nl/books?id=lFrcvQEACAAJ
29. Steen, A., Benzmüller, C.: The higher-order prover leo-iii. In: IJCAR. Lecture Notes in Computer Science, vol. 10900, pp. 108–116. Springer (2018)
30. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: IJCAR. Lecture Notes in Computer Science, vol. 8562, pp. 367–373. Springer (2014)
31. Sutcliffe, G.: The CADE ATP System Competition - CASC. AI Magazine **37**(2), 99–101 (2016)
32. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017)
33. Turner, D.A.: Another algorithm for bracket abstraction. J. Symb. Log. **44**(2), 267–270 (1979)
34. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 696–710. Springer (2014)
35. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. In: FSCD. LIPIcs, vol. 167, pp. 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
36. Vukmirovic, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: TACAS (1). Lecture Notes in Computer Science, vol. 11427, pp. 192–210. Springer (2019)
37. Waldmann, U., Tourret, S., Robillard, S., Blanchette, J.: A comprehensive framework for saturation theorem proving. In: IJCAR (1). Lecture Notes in Computer Science, vol. 12166, pp. 316–334. Springer (2020)